Type-Based Enforcement of Secure Programming Guidelines

Code Injection Prevention at SAP

Robert Grabowski[1] Martin Hofmann[1] Keqin Li[2]

[1]Ludwig-Maximilians-Universität München
[2]SAP Research Sophia-Antipolis

FAST Workshop · Leuven, Belgium · September 15-16, 2011

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");
output("<script>");
output("  alert('" + name + "');");
output("</script>");
```

- `name` may close ' ' quotes of `alert`, include malicious code

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");
output("<script>");
output("  alert(''); form.submit('http://...');");
output("</script>");
```

- name may close ' ' quotes of alert, include malicious code

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");
output("<script>");
output("  alert(''); form.submit('http://...');");
output("</script>");
```

- name may close ' ' quotes of alert, include malicious code

very common vulnerability, comes in many forms

Countermeasures Available to the Programmer

static program analysis

- dataflow analysis, information flow analysis, ...

$\rightarrow$ may be hard to understand and use; tools too specialized

runtime protection

- "tainted" tags, no cross-domain execution, ...

$\rightarrow$ very interpreter-dependent; runtime overhead

most common in practice: programming guideline

- easy to understand "best practice" for safe programming

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");
output("<script> alert('" + name + "'); </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings that are embedded in the output

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");
output("<script> alert('" +escapeToJs(name)+ "'); </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings
  that are embedded in the output

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");
output("<script> alert('" +escapeToJs(name)+ "'); </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings
  that are embedded in the output

obvious problem: programmer is responsible for following guideline

Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a Java programmer follows a given programming guideline.

Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a Java programmer follows a given programming guideline .

- types: familiar to programmers; extension of class types

Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a Java programmer follows a given programming guideline.

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code

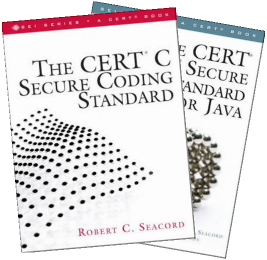
Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a Java programmer follows a given programming guideline.

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code
- guideline: correct usage of SAP sanitization framework

Vision: Programming Guideline Adherence Checking

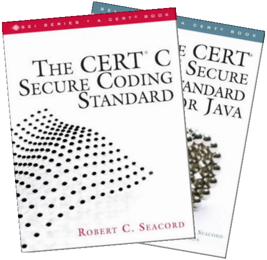
programming guidelines



- condensed expert knowledge, easy to apply for programmer
- cover many topics: memory safety, scheduling safety, resource handling, ...
- often described informally

→ ubiquitous in practice, but rarely research focus

Vision: Programming Guideline Adherence Checking



programming guidelines

- condensed expert knowledge,
  easy to apply for programmer
- cover many topics: memory safety,
  scheduling safety, resource handling, ...
- often described informally

$\rightarrow$ ubiquitous in practice, but rarely research focus

automatic support for implementing guidelines

- pick and formalize existing programming guideline
- provide easy-to-use tools that check adherence to guideline
- separate goal:
  evaluate guideline with respect to security properties

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output("  alert('" + escapeToJs(input) + "');");
  output("</script>");

}
```

ensure use of escape methods, depending on embedding position

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output("  alert('" + escapeToJs(input) + "');");
  output("</script>");

}
```

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output("  alert('" + escapeToJs(input) + "');");
  output("</script>");

}
```

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output("  alert('" + escapeToJs(input) + "');");
  output("</script>");

}
```

C1: sanitized for HTML

Literal: trusted

Literal

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);

  // case 2: JavaScript embedding
  output("<script>");
  output("  alert('" + escapeToJs(input) + "');");
  output("</script>");
```

C1: sanitized for HTML

Literal: trusted

Literal

<**script**>: enters JS mode

C2: sanitized for JavaScript

</**script**>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output(" alert('" + escapeToJs(input) + "');");
  output("</script>");
```

Input: untrusted

C1: sanitized for HTML

Literal: trusted

Literal

<script>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

</**script**>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output(" alert('" + escapeToJs(input) + "');");
  output("</script>");
```

Input: untrusted

C1: sanitized for HTML

Literal: trusted

Literal

<**script**>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

</**script**>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents
- calls to output induce abstract output sequence

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);


  // case 2: JavaScript embedding
  output("<script>");
  output(" alert('" + escapeToJs(input) + "');");
  output("</script>");
```

Input: untrusted

C1: sanitized for HTML

Literal: trusted

Literal

<**script**>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

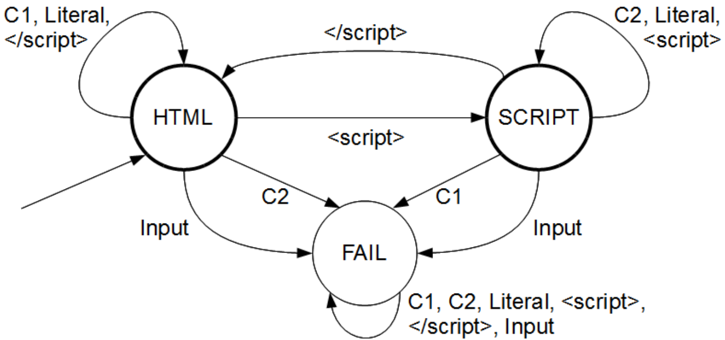
</**script**>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents
- calls to output induce abstract output sequence
- define set of allowed output sequences as regular language

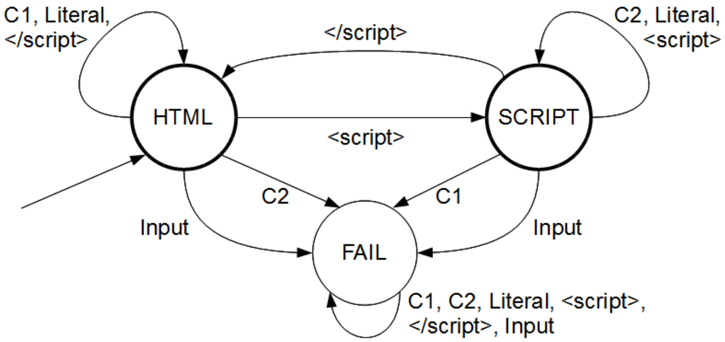
Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet $\Sigma =$ classifications



Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet Σ = classifications



- accepted words = allowed program output
- type system is defined with respect to given automaton

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	**SCRIPT**	**FAIL**
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	**SCRIPT**	**FAIL**
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

- $[w]$ is allowed if $\delta(q_0, w) \in F$

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	**SCRIPT**	**FAIL**
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

- $[w]$ is allowed if $\delta(q_0, w) \in F$
- class concatenation: $[w_1][w_2] = [w_1 w_2]$

String analysis for FJEUS

- FJEUS:
  Featherweight Java with Extended Updates and Strings

$$
\begin{aligned}
e \quad ::= \quad & x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \mid \texttt{if } x_1 = x_2 \texttt{ then } e_1 \texttt{ else } e_2 \mid \\
& \texttt{null} \mid \texttt{new } C \mid x.f \mid x_1.f{:=}x_2 \mid x.m(\overline{x}) \mid \\
& "str" \mid x_1 + x_2
\end{aligned}
$$

String analysis for FJEUS

- FJEUS:
  Featherweight Java with Extended Updates and Strings

$$
\begin{aligned}
e \quad ::= \quad & x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ x_1 = x_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& \texttt{null} \mid \texttt{new}\ C \mid x.f \mid x_1.f{:=}x_2 \mid x.m(\overline{x}) \mid \\
& "str" \mid x_1 + x_2
\end{aligned}
$$

- operational semantics with output trace / effect

$$(s, h) \vdash e \Downarrow v, h' \,\&\, w_t$$

- values v: object location
- traces w_t: word over classifications alphabet Σ

Instrumented string semantics

expression		**value**	**trace**
"abc"	$\Downarrow$	$\boxed{\dfrac{\texttt{String}}{\texttt{"abc"}\ \textit{Literal}}}$	& ϵ

Instrumented string semantics

expression		**value**	**trace**
"abc"	$\Downarrow$	$\frac{\texttt{String}}{\text{"abc"}}$ *Literal*	& ϵ

immutable string value...

...tagged with word over classification alphabet

Instrumented string semantics

	expression		value	trace
	"abc"	$\Downarrow$	$\boxed{\begin{array}{c}\texttt{String}\\ \hline \text{"abc"}\\ \mathit{Literal}\end{array}}$	$\&\ \epsilon$
$\boxed{\begin{array}{c}\texttt{String}\\ \hline \text{"abc"}\\ \mathit{Literal}\end{array}}$ + $\boxed{\begin{array}{c}\texttt{String}\\ \hline \text{"def"}\\ C1\end{array}}$		$\Downarrow$	$\boxed{\begin{array}{c}\texttt{String}\\ \hline \text{"abcdef"}\\ \mathit{Literal}\cdot C1\end{array}}$	$\&\ \epsilon$
	$\text{output}\left(\boxed{\begin{array}{c}\texttt{String}\\ \hline \text{"abcdef"}\\ \mathit{Literal}\cdot C1\end{array}}\right)$	$\Downarrow$	$\texttt{null}$	$\&\ \mathit{Literal}\cdot C1$

Type and effect system

FJ type system refined with annotated string types:

- defines typing judgement: $\boxed{\Gamma \vdash e : \tau \,\&\, U_t}$
- type $\tau \in \texttt{String}_U, \texttt{Object}, \texttt{Integer}, \ldots$
- annotations U, U_t are sets of classes [w]

Type and effect system

FJ type system refined with annotated string types:

- defines typing judgement: $\boxed{\Gamma \vdash e : \tau \,\&\, U_t}$
- type $\tau \in \mathtt{String}_U, \mathtt{Object}, \mathtt{Integer}, \ldots$
- annotations U, U_t are sets of classes $[w]$

Interpretation

If $(s, h) \vdash e \Downarrow v, h' \,\&\, w_t$ and $\Gamma \vdash e : \mathtt{String}_U \,\&\, U_t$

...and a couple of other premises...

then

- v points to string object tagged with w such that $[w] \in U$
- $[w_t] \in U_t$

Type and effect system

- typing rules approximate instrumented semantics:

$$\frac{\texttt{String}}{\text{"abc"}} \quad : \quad \texttt{String}_{\{[C1],[C2]\}}$$

$C1$

Type and effect system

- typing rules approximate instrumented semantics:

$$\boxed{\begin{array}{c}\texttt{String} \\ \hline \text{"abc"} \\ C1\end{array}} \quad : \quad \texttt{String}_{\{[C1],[C2]\}}$$

- external functions are assigned a trusted signature:

$$\mathit{getInputParameter}() : \texttt{String}_{\{[\mathit{Input}]\}} \;\&\; \{[\epsilon]\}$$

$$\mathit{escapeToHtml}(x : \texttt{String}_{\{[\mathit{Input}]\}}) : \texttt{String}_{\{[C1]\}} \;\&\; \{[\epsilon]\}$$

$$\mathit{output}(x : \texttt{String}_U) : \mathit{Void} \;\&\; U \quad \text{for all } U \subseteq M$$

Example: rejected program

$main\ () : Void =$

$\texttt{let}\ u = getInputParameter()\ \texttt{in}$

$\texttt{let}\ e = escapeToJs(u)\ \texttt{in}$

$output(e)$

Example: rejected program

$main\ () : Void =$

$\texttt{let}\ u = getInputParameter()\ \texttt{in}$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$\texttt{let}\ e = escapeToJs(u)\ \texttt{in}$

$output(e)$

Example: rejected program

$main\ () : Void =$

$\texttt{let}\ u = getInputParameter()\ \texttt{in}$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$\texttt{let}\ e = escapeToJs(u)\ \texttt{in}$

effect $= \{[\epsilon]\}$

$\Gamma(e) = \texttt{String}_{\{[C2]\}}$

$output(e)$

Example: rejected program

$main\ () : Void =$

$\quad$ `let` $u = getInputParameter()$ `in`

effect $= \{[\epsilon]\}$

$\Gamma(u) = \mathtt{String}_{\{[Input]\}}$

$\quad$ `let` $e = escapeToJs(u)$ `in`

effect $= \{[\epsilon]\}$

$\Gamma(e) = \mathtt{String}_{\{[C2]\}}$

$\quad output(e)$

effect $= \{[C2]\}$

Example: rejected program

$main\ () : Void =$

$\quad \texttt{let}\ u = getInputParameter()\ \texttt{in}$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$\quad \texttt{let}\ e = escapeToJs(u)\ \texttt{in}$

effect $= \{[\epsilon]\}$

$\Gamma(e) = \texttt{String}_{\{[C2]\}}$

$\quad output(e)$

effect $= \{[C2]\}$

overall effect: $[\epsilon][\epsilon][C2] = [C2]$

$\longrightarrow$ effect is not allowed (leads to FAIL state)

Inference of string type annotations and effects

inference algorithm based on declarative type system

- for each string variable and effect, create type variable U
- typing rules generate set constraints; solved externally

improved precision by context-sensitivity: [Beringer, G., Hofmann '10]

- polymorphic method types used for different call contexts
- type system is parametric in chosen context abstraction

Implementation

based on Java compiler implemented in Ocaml [Tse & Zdancewic '05]

specify refined string types and method effects for framework API

```
interface API {
  @ST("Input") String getInputParameter();
     @ST("C1") String escapeToHtml(@ST("Input") String str);
     @SE("C1") void   output(@ST("C1") String str);
}
```

no annotations required for actual code; all types and effects are inferred

```
  String s = api.escapeToHtml(input);
```

inference: set constraints as Datalog rules; solved with Succinct Solver [Nielson, Seidl, Nielson '03]

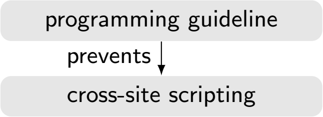
online demo at http://jsa.tcs.ifi.lmu.de/

Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

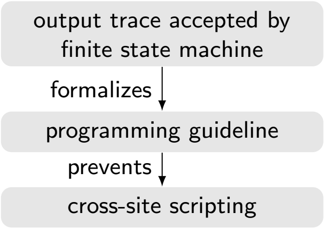
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



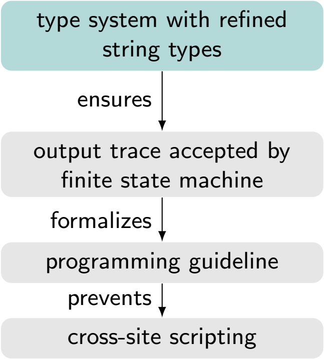
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



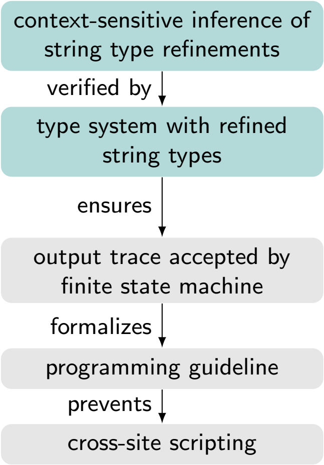
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

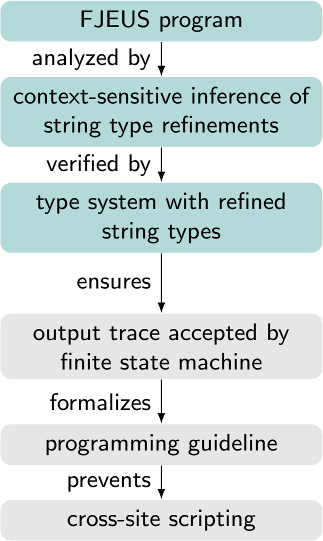


Summary

type-based enforcement of string handling guidelines to prevent XSS attacks

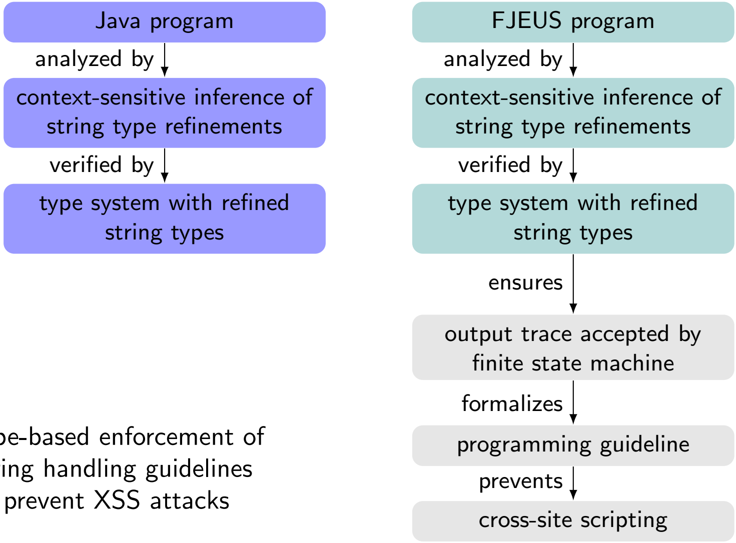


Summary



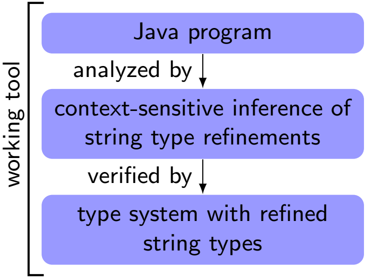
type-based enforcement of string handling guidelines to prevent XSS attacks

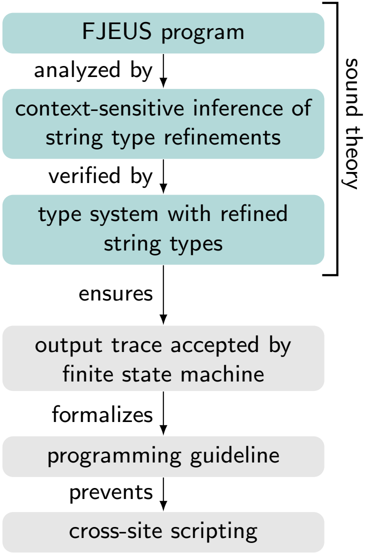
Summary



type-based enforcement of string handling guidelines to prevent XSS attacks

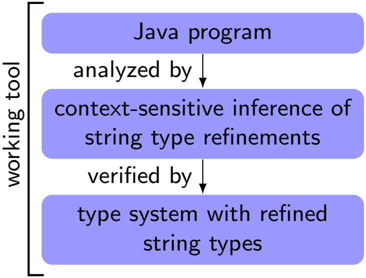
Summary

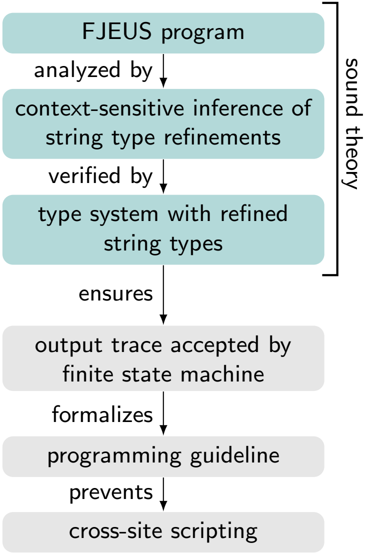




type-based enforcement of
string handling guidelines
to prevent XSS attacks

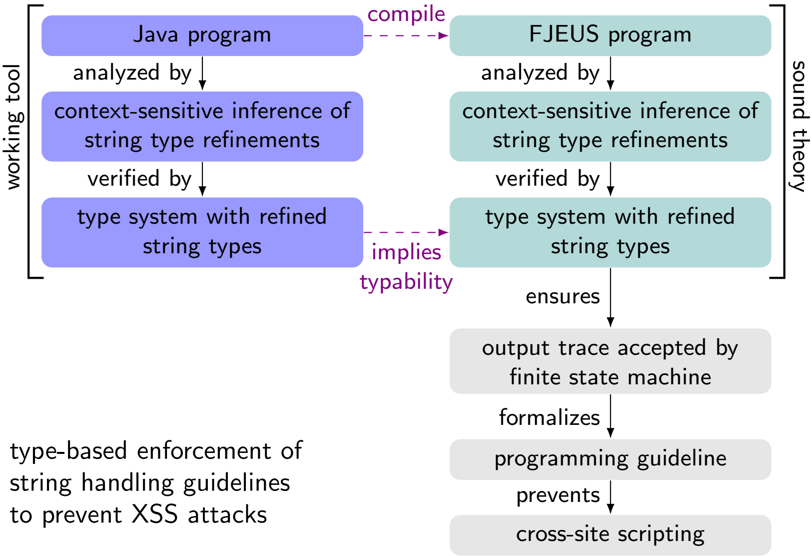
Summary and Future Work





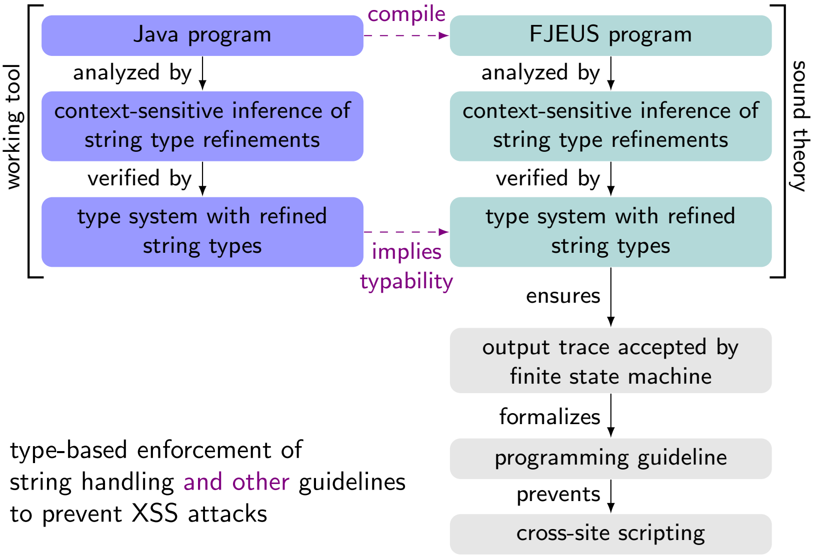
type-based enforcement of
string handling guidelines
to prevent XSS attacks

Summary and Future Work



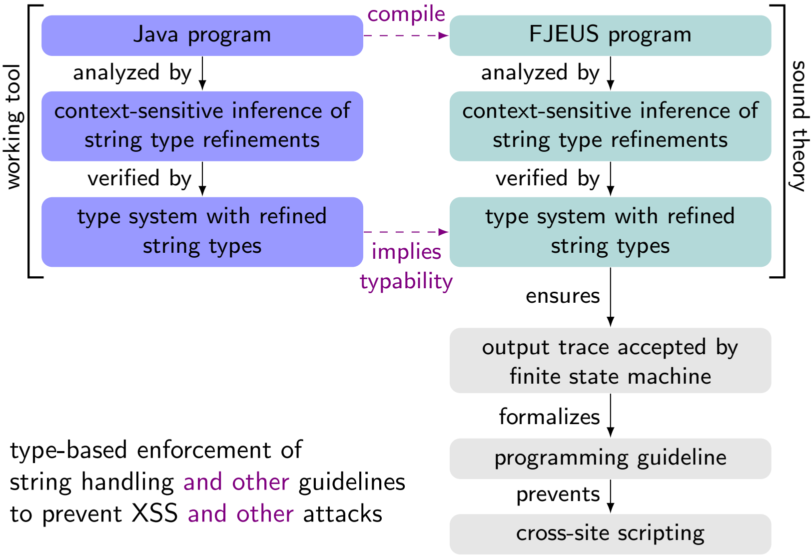
type-based enforcement of string handling guidelines to prevent XSS attacks

Summary and Future Work



type-based enforcement of
string handling and other guidelines
to prevent XSS attacks

Summary and Future Work



type-based enforcement of
string handling and other guidelines
to prevent XSS and other attacks

Thank you for your attention!

http://jsa.tcs.ifi.lmu.de/

Backup slides: Actual use cases

1. embed between HTML tags:

```
output("<h1> Hello " + escapeToHtml(x) + "</h1>");
```

2. embed as attribute value:

```
output("<img alt='" + escapeToAttr(x) + "' />");
```

3. embed as URL attribute value:

```
output("<img src='http://" + escapeToUrl(x) + "' />");
```

4. embed within JavaScript:

```
output("<script>");
output("  alert('" + escapeToJs(x) + "');");
output("</script>");
```

Instrumented string semantics (extract)

$$\frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \ \&\ w_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \ \&\ w_2}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2 \ \&\ w_1 \cdot w_2}$$

$$\frac{l \notin dom(h) \qquad h' = h[l \mapsto (Script, \text{"}\langle\text{script}\rangle\text{"})]}{(s,h) \vdash \text{"}\langle\text{script}\rangle\text{"} \Downarrow l, h' \ \&\ \epsilon}$$

$$\frac{h(s(x_1)) = (w_1, str_1) \qquad h(s(x_2)) = (w_2, str_2) \qquad l \notin dom(h) \qquad h' = h[l \mapsto (w_1 \cdot w_2, str_1 \cdot str_2)]}{(s,h) \vdash x_1 + x_2 \Downarrow l, h' \ \&\ \epsilon}$$

$$\frac{h(s(x)) = (w, \_)}{(s,h) \vdash output(x) \Downarrow \texttt{null}, h \ \&\ w}$$

Typing rules (extract)

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}}$$

$$\frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

$$\frac{\Gamma \vdash e : \texttt{String}_U \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

$$\frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\epsilon]\}}$$

$$\frac{\Gamma(x) = \texttt{String}_U}{\Gamma \vdash \texttt{output}(x) : \mathit{Void} \,\&\, U}$$

$$\frac{\Gamma \vdash e_1 : \tau \,\&\, V \qquad \Gamma, x : \tau \vdash e_2 : \tau' \,\&\, V'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau' \,\&\, VV'}$$

Improving precision of the analysis

building on earlier work [Beringer, G., Hofmann '10]

1. context-sensitive method analysis
2. class types refined with regions

Context-sensitive method analysis

```
String appendLn(String s) { return s + "\n"; }
```

```
String x = appendLn("Your name is:");
String y = appendLn(sanitizedInput);
```

$$appendLn : \mathtt{String}_{\{[\epsilon],[C1]\}} \to \mathtt{String}_{\{[\epsilon],[C1]\}}$$

Context-sensitive method analysis

```
String appendLn(String s) { return s + "\n"; }
```

```
16 String x = appendLn("Your name is:");
17 String y = appendLn(sanitizedInput);
```

- use different method types for different calls of the method

call site	*type*
16	$appendLn : \mathtt{String}_{[\epsilon]} \to \mathtt{String}_{[\epsilon]}$
17	$appendLn : \mathtt{String}_{[C1]} \to \mathtt{String}_{[C1]}$

Context-sensitive method analysis

```
15  String doSomething(String sanitizedInput) {
16    String x = appendLn("Your name is:");
17    String y = appendLn(sanitizedInput);
18  }
```

```
54  doSomething(escapeToHtml(input));
```

```
80  doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack	*type*
16 :: 54	$appendLn : \texttt{String}_{[\epsilon]} \rightarrow \texttt{String}_{[\epsilon]}$
17 :: 54	$appendLn : \texttt{String}_{[C1]} \rightarrow \texttt{String}_{[C1]}$

Context-sensitive method analysis

```
15  String doSomething(String sanitizedInput) {
16    String x = appendLn("Your name is:");
17    String y = appendLn(sanitizedInput);
18  }
```

```
54  doSomething(escapeToHtml(input));
```

```
80  doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack	*type*
16 :: 54	$\mathit{appendLn} : \mathtt{String}_{[\epsilon]} \to \mathtt{String}_{[\epsilon]}$
17 :: 54	$\mathit{appendLn} : \mathtt{String}_{[C1]} \to \mathtt{String}_{[C1]}$
16 :: 80	$\mathit{appendLn} : \mathtt{String}_{[\epsilon]} \to \mathtt{String}_{[\epsilon]}$
17 :: 80	$\mathit{appendLn} : \mathtt{String}_{[C2]} \to \mathtt{String}_{[C2]}$

Context-sensitive method analysis

```
String doSomething(String sanitizedInput) {
  String x = appendLn("Your name is:");
  String y = appendLn(sanitizedInput);
}
```

```
doSomething(escapeToHtml(input));
```

```
doSomething(escapeToJs(input));
```

- distinguish method types by call context from finite set *Cxt*
- method call rule relies on context switch function

$$\phi : Cxt \times Cls \times Mtd \times PP \rightarrow Cxt$$

  to select the method type for the invoked method
- inference generates new method type for unseen contexts

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

increases typing precision:

$$fty(\texttt{List}, r, \texttt{elem}) = \texttt{String}_{\{[\mathit{Input}]\}} \quad fty(\texttt{List}, s, \texttt{elem}) = \texttt{String}_{\{[C1]\}}$$
$$fty(\texttt{List}, r, \texttt{next}) = \texttt{List}_{\{r\}} \quad fty(\texttt{List}, s, \texttt{next}) = \texttt{List}_{\{s\}}$$

Region type system

- type system parametrized with abstraction principles from pointer analysis
  - choice of regions for new objects
  - contexts
- inference: follows existing analysis [Whaley & Lam '04]
  - generate constraints for points-to relations as Datalog rules
  - use external, highly optimized solver
  - interprete resulting relations as types and verify them with type system
- for more information: see LPAR paper